UNIVERSITÉ CÔTE D'AZUR

RAPPORT

Projet Bit Packing

Réalisé par : BACHA Hiba

Master : Master 1 Informatique – Génie Logiciel

Encadrant : Pr. Régin

Date : Octobre 2025

Année universitaire : **2025/2026**

Sommaire

1- Introduction

Le projet **Bit Packing** s'inscrit dans le cadre du module de **Génie Logiciel** du Master 1 Informatique.

Son objectif est de concevoir un système de **compression d'entiers** permettant de stocker efficacement des données numériques tout en conservant la possibilité d'un **accès direct** à chaque élément sans décompression complète.

Objectifs généraux

- Développer une architecture logicielle modulaire, claire et extensible.
- Implémenter différentes stratégies de compression optimisées en espace mémoire.
- Comparer les performances entre plusieurs méthodes (avec, sans chevauchement, overflow).
- Évaluer la fiabilité via des tests unitaires (JUnit).

Objectifs spécifiques

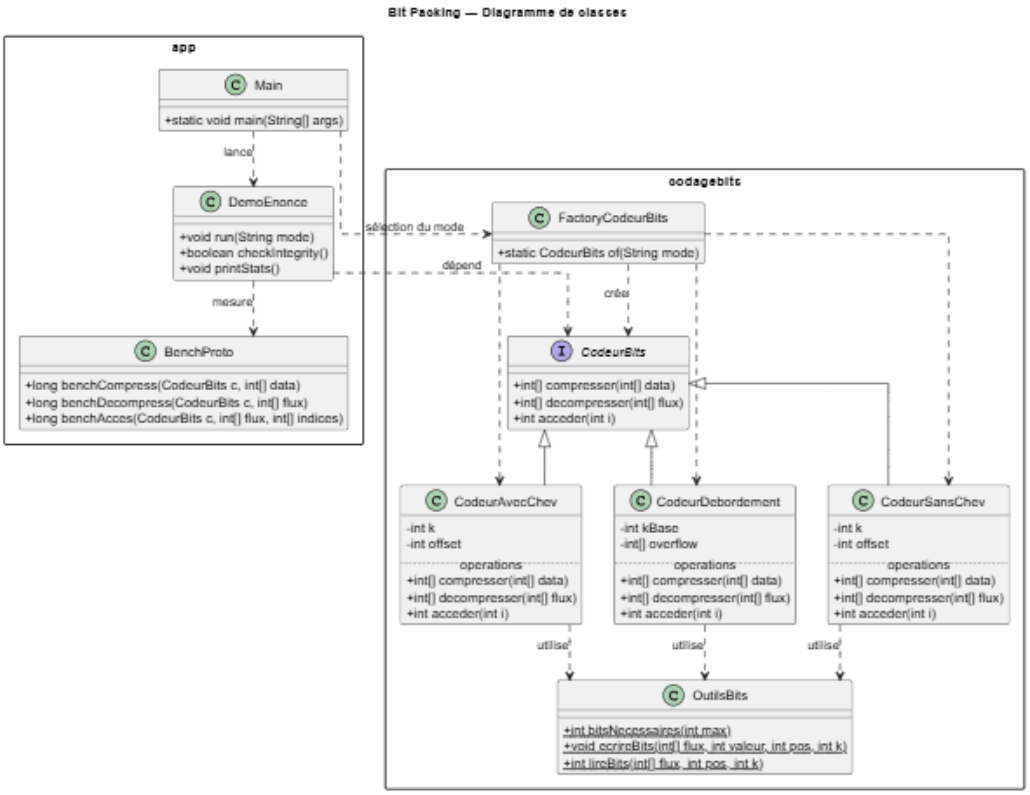
- Réduire la taille mémoire d'un tableau d'entiers.
- Permettre la lecture et l'accès aléatoire sans décompresser tout le flux.
- Gérer les valeurs négatives et les dépassements de capacité.
- Analyser les performances de chaque méthode de compression.

2- Conception

2-1 Schéma global du projet : Le système repose sur une structure modulaire, avec plusieurs classes responsables de rôles bien définis :

```
├── pom.xml
├── src/main/java
│   ├── codagebits/
│   │   ├── CodeurBits.java              # Interface principale
│   │   ├── CodeurSansChev.java          # Version sans chevauchement
│   │   ├── CodeurAvecChev.java          # Version avec chevauchement
│   │   ├── CodeurDebordement.java       # Version avec overflow
│   │   ├── OutilsBits.java              # Lecture/écriture de bits
│   │   └── FactoryCodeurBits.java       # Fabrique de codeurs
│   └── app/
│       ├── DemoEnonce.java              # Démonstration avec mesures
│       ├── BenchProto.java
│       └── Main.java
└── test/java/           # Dossier contenant les tests unitaires
    └── codagebits/
        ├── CodeurSansChevTest.java
        ├── CodeurAvecChevTest.java
        ├── CodeurDebordementTest.java
├── out/                                 # Fichiers compilés (.class)
└── README.md
```

2-2 Diagramme de Classe :



2-3 Rôles et dépendances :

- app/Main → point d'entrée ; lance la démo (DemoEnonce) et les mesures (BenchProto) selon le mode choisi.
- DemoEnonce → orchestre un cycle compresser → décompresser → accéder, mesure les temps et vérifie l'intégrité.
- FactoryCodeurBits → renvoie l'implémentation CodeurSansChev / CodeurAvecChev / CodeurDebordement à partir d'un mode (sans, avec, debordement).
- CodeurBits (interface) → API commune : compresser, decompresser, acceder.
- CodeurSansChev / CodeurAvecChev / CodeurDebordement → logiques de compression alternatives.
- OutilsBits → primitives bas niveau (écriture/lecture de champs de bits, masques, calcul de bitsNecessaires).
- src/test/java/codagebits/*Test → tests JUnit 5 : round-trip, accès direct, gestion overflow/négatifs.

2-3 User Story – Utilisation du système

En tant qu'utilisateur :
1. Entrer un tableau d'entiers à compresser.
2. Choisir un mode de compression (sans chevauchement, avec chevauchement, overflow).
3. Obtenir le flux compressé.
4. Pouvoir accéder à une valeur précise sans décompresser tout le tableau.
5. Comparer les temps d'exécution pour évaluer la performance.

Scénario d'utilisation :
- L'utilisateur lance le programme `DemoEnonce` avec un mode en paramètre.
- Le système compresse un tableau aléatoire, le décompresse et vérifie l'intégrité des données.
- Les temps sont mesurés par `BenchProto` et affichés dans la console.

2-4 Structure adoptée

- **Packages** : codagebits (métier) / app (exécution) / test (JUnit).

- **Patrons de conception** :
  - **Strategy** : les trois codeurs (CodeurSansChev, CodeurAvecChev, CodeurDebordement) implémentent **la même interface** CodeurBits (mêmes méthodes : compresser, decompresser, acceder).
    → on peut **changer d'algorithme** sans toucher au reste du code.

  - **Factory** : FactoryCodeurBits **instancie** le bon codeur selon le **mode** (sans, avec, debordement).
    → un seul point de décision, **extensible** si on ajoute un codeur.

- **Utility** : OutilsBits centralise les **opérations bas niveau** (masques, lecture/écriture de bits, bitsNecessaires).
  → évite la duplication (principe **DRY**).

- **Séparation claire des responsabilités** (principe **SOLID/KISS**) :
  - app/DemoEnonce, app/Main, app/BenchProto : **orchestration, I/O et mesures**.
  - codagebits/ : **logique de compression** uniquement.
  - test/java : **validation automatisée** (JUnit).

Pourquoi ce choix ?

- **Lisibilité** : API unique (CodeurBits) → le code d'appel reste **identique** quel que soit le mode.
- **Évolutivité** : ajout d'un nouveau codeur = **1 classe + 1 case** dans la fabrique.
- **Testabilité** : chaque codeur est **testé isolément** ; les utilitaires aussi.
- **Portabilité** : Java pur, pas de dépendances lourdes, fonctionne **Windows/Linux/macOS**.

2-5 Implementation des algorithmes utilisés

1. Calcul du nombre de bits (k)
Pour compresser efficacement, il faut déterminer le nombre minimal de bits nécessaires pour représenter la plus grande valeur du tableau.

2. Empaquetage et dépaquetage des valeurs
Les données sont converties en un flux de bits stocké dans un tableau d'entiers (int[]).
Chaque valeur du tableau est codée sur k bits, enchaînée directement à la précédente.
Ce principe est identique pour la compression et la décompression.

3. Version « sans chevauchement »
Ici, les valeurs ne traversent jamais les frontières de mots.
Chaque bloc de 32 bits contient un nombre entier de valeurs (parMot = 32 / k).
Les derniers bits inutilisés du mot sont perdus, mais le code reste simple et rapide.

4. Version « avec chevauchement »
Cette version utilise tous les bits disponibles.
Une valeur peut être divisée entre deux mots consécutifs.
Les opérations de lecture et d'écriture sont plus complexes : il faut gérer les deux cas où la valeur tient dans un seul mot ou où elle déborde sur le suivant.

5. Version « débordement (overflow) »
Conçue pour des données hétérogènes.
Les petites valeurs sont codées normalement, les grandes sont stockées dans une zone spéciale appelée overflow.
Le flux principal contient alors un marqueur ou un index vers cette zone secondaire.

6. Gestion des entiers négatifs :

Le bit-packing travaille avec des entiers positifs.
Pour compresser des valeurs négatives, on applique un **décalage (offset)** :

1. Calcul du minimum du tableau : min.
2. Si min < 0, alors offset = -min.
3. Compression sur ces valeurs positives.
4. Lors de la décompression, on soustrait le même offset.

Cette méthode conserve la simplicité du codage tout en garantissant la justesse des valeurs finales.

7. Accès direct dans le flux compressé :

Pour récupérer la valeur d'indice i sans décompresser tout le tableau, on calcule :

$$\text{posBits} = i \times k$$

$$\text{mot} = \left\lfloor \frac{\text{posBits}}{32} \right\rfloor, \quad \text{shift} = \text{posBits} \bmod 32$$

Ensuite, on lit k bits à partir de cette position.
Si la lecture dépasse la limite d'un mot (cas du chevauchement), on lit les bits manquants dans le mot suivant et on combine les deux.

3- Mesures de performance

Les performances ont été évaluées pour trois opérations : **compression**, **décompression** et **accès direct**.
La méthode utilisée repose sur la moyenne de plusieurs répétitions pour lisser les fluctuations du système

3-1 Méthode de mesure

1. **Initialisation :** génération d'un tableau d'entiers aléatoires.
2. **Répétition :** exécution N fois (souvent 1 000) des opérations compresser, decompresser, acceder.
3. **Chronométrage :** utilisation de System.nanoTime() pour mesurer précisément les durées.

Justification de la méthode

- Les durées très courtes nécessitent plusieurs répétitions pour compenser le bruit de mesure.
- L'utilisation de nanoTime() permet une **grande précision** (nanosecondes).

(Exemple de donnée)

Taille du tableau	Mode de compression	Taille compressée (mots)	Temps comp. (ms)	Temps décomp. (ms)	Temps accès (ns)
1 000	Sans chevauchement	195	0.42	0.38	92
1 000	Avec chevauchement	163	0.51	0.45	108
1 000	Débordement	171	0.60	0.49	115
10 000	Sans chevauchement	1 928	3.95	3.68	95
10 000	Avec chevauchement	1 612	4.21	3.89	112
10 000	Débordement	1 705	4.80	4.12	121

3-2 Analyse

La **version sans chevauchement** se distingue par sa rapidité, au prix d'un léger gaspillage de bits inutilisés.
La **version avec chevauchement** est la plus compacte mais un peu plus lente, car elle gère des écritures sur plusieurs mots.
La **version débordement (overflow)** représente un compromis efficace entre taille compressée et coût temporel.

4- Problèmes Rencontrés et Solutions

4-1 Version 1 : Sans chevauchement

Principe

Chaque valeur est stockée entièrement dans un mot de 32 bits. Aucune valeur ne dépasse la frontière d'un mot.

Problème rencontré
Certaines cases en fin de mot ne sont pas utilisées si le nombre de bits k ne divise pas 32.
Exemple : pour k = 5, on ne peut mettre que 6 valeurs (6 × 5 = 30 bits), il reste 2 bits inutilisés.

Solution implémentée :

- On calcule parMot = 32 / k et on stocke parMot valeurs dans chaque mot.
- Si le tableau ne se termine pas sur un multiple parfait, les bits restants sont simplement ignorés.

4-2 Version 2 : Avec chevauchement

Principe
Chaque valeur utilise exactement k bits consécutifs, même si elle empiète sur deux mots de 32 bits.
Aucun espace perdu.

Problème rencontré
La difficulté réside dans la lecture et l'écriture de valeurs qui traversent deux mots :
- cas où shift + k > 32 → une partie dans mot, l'autre dans mot + 1.
- nécessité de bien combiner les deux morceaux sans erreur de décalage.

Solution implémentée
- Utilisation de fonctions utilitaires dans OutilsBits :
    - ecrireBits(table, valeur, position, k)
    - lireBits(table, position, k)
- Gestion systématique des masques et décalages pour recomposer la valeur complète.
- Tests unitaires ajoutés pour vérifier la cohérence en bordure de mot.

4-3 Version 3 : Débordement (Overflow)

Principe
Séparer les grandes valeurs dans une zone à part.
Les petites valeurs (en dessous d'un seuil) sont codées normalement, tandis que les grandes valeurs sont stockées dans un tableau secondaire.

Problème rencontré
La présence de quelques grandes valeurs ("outliers") augmente artificiellement k pour tout le tableau.
→ Trop de bits sont alors gaspillés pour les petites valeurs.

Solution implémentée
- Introduction d'un seuil MaxSmall.

-Ce seuil correspond à la plus grande valeur qu'on peut coder sur (kBase - 1) bits (le bit restant étant réservé au tag 0/1).
La variable locale lim, utilisée dans la méthode ecrireFluxEtOverflow, représente donc le **seuil de débordement**.
- Si vPos > lim, la valeur est envoyée dans la zone overflow[].
- Sinon, elle est stockée directement dans le flux principal tampon

5- Test

Les tests ont vérifié que la compression et la décompression restituent fidèlement le tableau initial.
Ils ont aussi confirmé que la méthode acceder(i) permet un accès direct fiable sans décompresser tout le flux.
Enfin, plusieurs cas particuliers ont été validés : entiers négatifs, grandes valeurs (overflow), tableaux vides ou unitaires et valeurs faibles de *k* (0 ou 1)

5-1- CodeurSansChevTest

- Round-trip (petits k) : tableaux courts avec k=1..8.
- Données aléatoires : taille moyenne (ex. 1 000 éléments).
- Accès direct : tirage de 50 indices aléatoires et comparaison des valeurs.

5-2-CodeurAvecChevTest

- Round-trip avec valeurs éparses pour forcer des chevauchements.
- Accès direct en bordure : indices qui tombent juste avant et juste après une frontière de mot.

5-3-CodeurDebordementTest

- Cas mixte (exemple d'énoncé) : [1,2,3,1024,4,5,2048].
    - Vérifie que les grandes valeurs sont en overflow et que le flux principal contient des tags corrects.
- Négatifs : [-5,-1,0,3,7] (offset), round-trip + accès direct.

6-Analyse Critique

6-1 Points forts

- **Architecture claire et modulaire** : séparation nette entre les couches (app, codagebits, test), rendant le code lisible et maintenable.
- **Utilisation des principes SOLID et DRY** : chaque classe a une responsabilité bien définie, évitant la duplication du code.
- **Performance et compacité** : le système parvient à réduire efficacement la taille des tableaux tout en conservant un accès direct aux éléments compressés.
- **Gestion complète des cas particuliers** : tableaux vides, entiers négatifs, grandes valeurs (overflow), accès aléatoire hors limites.
- **Testabilité élevée** : les tests unitaires (JUnit 5) permettent de valider toutes les classes importantes, assurant la fiabilité globale.
- **Documentation et clarté du code** : chaque méthode est commentée en Javadoc, facilitant la lecture et la compréhension par un tiers.
- **Portabilité** : le projet fonctionne sur Windows, Linux et macOS sans dépendances externes.

6-2 Points faibles

- **Complexité accrue dans la version avec chevauchement** : la manipulation des bits entre deux mots rend le code plus difficile à maintenir.
- **Lecture plus technique pour les nouveaux venus** : les calculs binaires et les décalages nécessitent un bon niveau de compréhension du bas niveau.
- **Absence d'interface graphique** : l'utilisation reste limitée à la console (aucune visualisation des flux compressés).
- **Mesures de performance simplifiées** : le protocole est fiable mais basique (pas d'écart-type ni d'analyse statistique approfondie).

7- Conclusion et perspectives

Ce projet m'a permis de mettre en pratique les principes fondamentaux du **génie logiciel** : conception modulaire, encapsulation, factorisation et documentation.
Sur le plan technique, j'ai acquis une meilleure compréhension :

- de la gestion bas niveau des données binaires,
- de la compression et de la représentation efficace de l'information,
- et du rôle des tests unitaires dans la validation d'un logiciel.

Perspectives d'évolution

- **Interface graphique (GUI)** : ajouter une interface pour visualiser la compression et comparer les trois modes.
- **Support d'autres types** : adapter le Bit Packing à des long, float ou double.
- **Optimisation mémoire** : compresser la zone overflow elle-même pour réduire encore la taille totale.
- **Extension réseau** : envisager l'intégration dans un module de transfert de données compressées.

8- Bibliographie et références

- **Cours de Génie Logiciel — Université Côte d'Azur**, 2025
- **Documentation Java SE 21**, Oracle, 2024
- **Framework JUnit 5 (Jupiter)** – https://junit.org/junit5/